ulm university universität ulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Evaluation von Hybrid Mobile App Frameworks im Kontext von medizinischen Applikationen

Bachelorarbeitan der Universität Ulm

Vorgelegt von:
Julius Friedrich
julius.friedrich@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Marc Schickler

2017

Fassung 18. Januar 2017

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Medizinisch unterstützende Applikationen sind Anwendungen, die einen Teil der ärztlichen Behandlung von Patienten, sowie auch deren Betreuung unterstützen oder sogar abnehmen sollen. Neben den schon existierenden Anwendungsbereichen von mobilen Applikationen bietet sich auch dieser Anwendungsbereich durch die steigende Anzahl an Smartphone-Nutzern an. Diese Applikationen bergen jedoch auch Gefahren und Risiken für den Patienten. Um diese Gefahren und Risiken einschätzen und minimieren zu können, muss in diesem Gebiet geforscht werden. Dafür werden Applikationen benötigt, mit denen Tests und Studien durchgeführt werden können. Bei der Entwicklung solcher Applikationen, ist es wichtig, das richtige Framework zu wählen. Durch das Aufkommen neuer Mobile App Frameworks wird diese Entscheidung zunehmend schwieriger.
Ziel dieser Arbeit ist es zu untersuchen, welches Framework sich für das Erstellen von medizischen Applikationen am besten eignet. Dazu werden grundlegende Funktionen der Applikation mit den Frameworks Meteor, Ionic und Xamarin realisiert. Anschließend werden die entstandenen Applikationen analysiert und evaluiert. Zum Abschluss der Arbeit wird ein Fazit gezogen, in dem eine Empfehlung ausgesprochen wird, welches der Frameworks sich am besten zur Entwicklung der Applikation eignet.

Inhaltsverzeichnis

1

Einleitung

In der Medizin gibt es schon seit langer Zeit Software, die Ärzte bei ihrer Arbeit unterstützt. Beispielsweise braucht ein Ultraschallgerät entsprechende Software. Das Interesse an technischer Unterstützung ist groß, da sie arbeitserleichternd wirken kann. Mit der steigenden Anzahl von Smartphone-Benutzern bietet sich die Möglichkeit, medizinische Behandlung in Form von mobilen Applikationen zu unterstützen.
Medizinische Applikationen können verschiedene Vorteile mit sich bringen. Ein Arzt könnte mehrere Patienten durch Zeitersparnisse behandeln, Kommunikationswege könnten kürzer und bürokratischer Aufwand verringert werden. Trotzdem bieten Ferndiagnosen, Fehler bei der Datenübertragung und Fehler in der Software auch Gefahren, die erforscht werden müssen.
Das Institut für Datenbanken und Informationssysteme der Universität Ulm beschäftigt sich mit der Entwicklung von medizinisch unterstützenden Applikationen und möchte eine solche Applikation erschaffen, um dieses Gebiet weiter zu erforschen.
Der größte programmatische Aufwand in der Erstellung vieler Applikationen entsteht bei der Unterstützung von mehreren Plattformen. Meist wird eine Applikation für eine Plattform erstellt und dann für andere Plattformen nachgebaut. Dieses Vorgehen kann speziell in der Wartung sehr zeitaufwändig sein.
Einen anderen Ansatz bieten Hybride Frameworks, die mit nur einem Sourcecode arbeiten, der dann für die verschiedenen Plattformen aufbereitet wird. Dahinter steckt die Idee, Code nur einmal zu schreiben und auch nur einen Code zu warten. Diese Technologie ist jedoch relativ neu und es sind im Zeitraum der letzten Jahre viele neue Frameworks mit Fokus auf unterschiedliche Funktionalitäten entstanden.

1.1 Problemstellung

Beim Entwurf einer mobilen Applikation gilt es im Vorhinein verschiedene Vorkehrungen und Entscheidungen zu treffen. Neben der genauen Definition der funktionalen und nicht funktionalen Anforderungen, ist vor allem die Frage nach dem richtigen Werkzeug zu beantworten. Es muss also anhand der Anforderungen festgelegt werden, welches Framework, oder im weiteren Sinne, welche Programmiersprache gewählt wird.
Möglich sind zum einen native Applikationen, die speziell für ein Betriebssystem programmiert werden. Zum Beispiel eine native Android-Applikation für Mobilgeräte mit einem Android-Betriebssystem. Da der Fokus dieser Forschung auf dem medizinischen Aspekt liegen soll, ist es vorteilhaft, den technischen, beziehungsweise programmatischen Aufwand zu reduzieren. Man kann diesen Aufwand reduzieren, indem man sich für ein hybrides Framework entscheidet. Für diese medizinische Applikation gilt es nun ein Framework zu finden, das die Anforderungen bestmöglich erfüllen kann.

1.2 Zielsetzung

In dieser Arbeit sollen zunächst Frameworks, die für die Applikation in Frage kommen, ausgesucht werden. Anschließend sollen die genauen Anforderungen an die Applikation definiert werden. Aus diesen Anforderungen sollen Funktionen abgeleitet werden, die repräsentativ für die Anforderungen sind. Diese abgeleiteten Funktionen sollen mit Hilfe der verschiedenen Frameworks umgesetzt werden.
Anschließend werden die verschiedenen Umsetzungen der Funktionen analysiert und verglichen. Anhand dieser Erkenntnisse soll evaluiert werden, welches Framework sich am besten eignet, um abschließend eine fundierte Empfehlung für die reelle Umsetzung der Applikation abzugeben.

1.3 Struktur der Arbeit

Zunächst werden die funktionalen und nicht funktionalen Anforderungen an die Applikation in Kapitel 2 spezifiziert. In Kapitel 3 folgt eine Einführung in die grundlegenden Funktionsweisen der jeweiligen Frameworks. Im 4. Kapitel werden die umgesetzten Applikationen anhand der festgelegten Funktionen analysiert und verglichen. In Kapitel 5 werden Vor- und Nachteile der Frameworks aufgezeigt, die aus Kapitel 4 hervorgehen, aber auch vom jeweiligen Framework selbst bedingt sind. Abschließend wird in Kapitel 6 ein Fazit gezogen und damit eine Empfehlung für Framework abgegeben.

2

Allgemeine Grundlagen

In diesem Kapitel werden die Grundlagen beschrieben, die für das Verständnis dieser Arbeit erforderlich sind. Zum einen wird darauf eingegangen, wie eine REST-API und eine mobile Applikation kommunizieren können und was Cordova ist, beziehungsweise wie Cordova benutzt wird.

2.1 REST-Services

Representational **S**tate **T**ransfer wurde im Jahr 2000 von Roy Fielding in seiner Dissertation [1] als Architekturstil eines Webservices definiert. Webservices benutzen die REST-Architektur für die Datenübertragung zwischen Client und Webserver und für die Darstellung von Programmzuständen. Die Datenübertragung beim REST basiert auf dem HTTP-Protokoll. Die Architektur eines REST-Services ist in Abbildung 2.1 dargestellt.

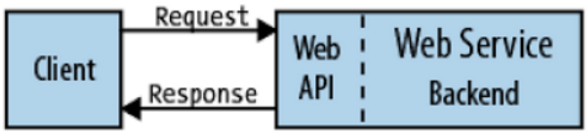

Abbildung 2.1: Architektur eines REST-Services. [2]

2.1.1 Funktionsweise

Webservices die eine REST-Architektur verwenden, bestehen aus einer Web-API und einem Backend, das von der API angesprochen werden kann. Ein simples Anwendungsszenario ist ein Client, der Ressourcen vom Webservice anfragt. Jede Ressource hat dabei einen eigenen eindeutigen Identifier, in Form eine URI (**U**niform **R**essource **I**dentifier), beziehungsweise im Fall eines Webservices haben Ressourcen eine URL (**U**niform **R**essource **L**ocator).
Ein einfaches Beispiel für eine URL ist in Listing 2.1 gegeben. Die Web-API stellt dabei dem Client eine Route zur Verfügung, in diesem Beispiel eine statische Route. Der Webservice kann nun die Anfrage (Request) des Clients mit einer Antwort (Response) in der die Ressource enthalten ist, beantworten.

```
1 example.com/ressource1
```

Listing 2.1: Beispiel für einen Uniform Ressource Identifier

Bei der Kommunikation mit dem Server kann der Client zwischen den verschiedenen HTTP-Methoden wählen. Bei der realen Verwendung kommen zumeist die GET-Methode, beispielsweise zum Anfragen von Ressourcen und die POST-Methode, beispielsweise zum Übertragen von Formulardaten zum Einsatz. Außerdem existieren auch die PUT- und die DELETE-Methode, die zum Hochladen, Ersetzen und Löschen von Ressourcen verwendet werden können. In den meisten Webservices werden PUT und DELETE aber nur selten verwendet, da das Verwalten von Ressourcen als sicherheitskritisch eingestuft wird und deshalb oftmals vom Backend geregelt wird.

Eine REST-API kann auch als Instrument verwendet werden, das komplexere Vorgänge im Backend des Webservices steuert. Zum Beispiel kann beim Senden von Daten an eine bestimmte URL der REST-API, ein Vorgang im Backend gestartet werden, der dann eine Datenbankoperation mit den erhaltenen Daten durchführt. Bei solch einer Übertragung von Daten ist es wichtig, dass Client und Server ein einheitliches Datenformat verwenden.

2.1.2 JSON-Format

Ein mögliches Format um einen einheitlichen und konsistenten Datenaustausch zu gewährleisten, ist das JSON-Format. JSON ist eine Abkürzung für **J**ava**S**cript **O**bject **N**otation und wurde in RFC 7195 [3] von Douglas Crockford in spezifiert. Das Format wird hauptsächlich bei der Kommunikation zwischen Anwendungen verwendet.

Die Datensätze werden in Textform abgespeichert und sind so auch für den Menschen leicht analysierbar sind. Ein wichtiger Vorteil ergibt sich daraus, dass JSON-Datensätze sehr kompakt sind und dadurch nur einen geringen Overhead an zusätzlichen Informationen haben, beziehungsweise brauchen. Dieser Vorteil ergibt sich nicht zuletzt aus der Strategie, *Key-Value*-Paare zu verwenden. Dabei ist es neben einfachen Datentypen wie Booleans oder Integern, möglich Arrays und komplexe Javascript Objekte im JSON-Format zu serialisieren. Ein Beispiel für ein JSON-Datensatz ist in Listing 2.2 gegeben, die genaue Syntax ist in RFC 7195 [3] nachzulesen.

```
1  {"house": {
2    "id": "1",
3    "height": "2.8",
4    "floorArea": "120",
5    "rooms": [
6        {"type": "kitchen", "floorArea": "30"},
7        {"type": "bathroom", "floorArea": "20"},
8        {"type": "livingroom", "floorArea": "70"}
9    ]
10 }}
```

Listing 2.2: Ein Beispiel für einen JSON-Datensatz

2.2 Cordova

Apache Cordova (früher PhoneGap) ist eine Open-Source Software, die es ermöglicht, mobile Applikationen auf der Basis von moderner Webtechnologie wie HTML5, CSS3 und Javascript zu programmieren. Cordova ist ein *Cross-Plattform-Framework*, das heißt, dass eine Cordova Applikation auf mehreren Betriebssystemen benutzt werden kann. Unterstützt werden die Betriebssysteme Android, iOS und Windows Phone.
Die resultierenden Applikationen sind sogenannte Hybride. Es sind auf der einen Seite keine Web-Applikationen, da sie nicht im gewöhnlichen mobilen Browser, sondern als eigene Applikation installiert sind. Außerdem haben Cordova Applikationen die Möglichkeit auf Hardware, wie beispielsweise auf die Kamera oder die GPS-Position zuzugreifen. Auf der anderen Seite sind es offensichtlich auch keine nativen Applikationen, da das gesamte Rendering in einem Webview (eine Art Browser-Fenster ohne erweiterte Bedienungsoberfläche) geschieht.

2.2.1 Funktionsweise

Eine Cordova Applikation besteht, wie in 2.2 gezeigt wird aus mehreren Modulen: der Web-Applikation selbst, einer WebView und aus verschiedenen Plugins. Die Web-Applikation, bestehend aus HTML, CSS und Javascript, enthält die Logik der Anwendung. Dieser Teil der Applikation funktioniert auch eigenständig in einem modernen Browser und gleicht der Form einer normalen Web-Applikation.

Die Web-Applikation läuft aber unter Cordova in einem modifizierten WebView, der im Gegensatz zu einem normalen mobilen Browser zusätzliche Schnittstellen bietet, um so auf spezielle Hardware des Mobilgerätes zuzugreifen. Für diese erweiterte Kommunikation mit der WebView stellt Cordova eine eigene API zur Verfügung.
Die Kommunikation mit der Hardware funktioniert aber nicht über direktem Wege mit der Hardware-Komponente, sondern über die Cordova Plugins. Jede Hardware-Komponente wird über ein passendes Plugin angesprochen. Diese Plugins enthalten nativen Code

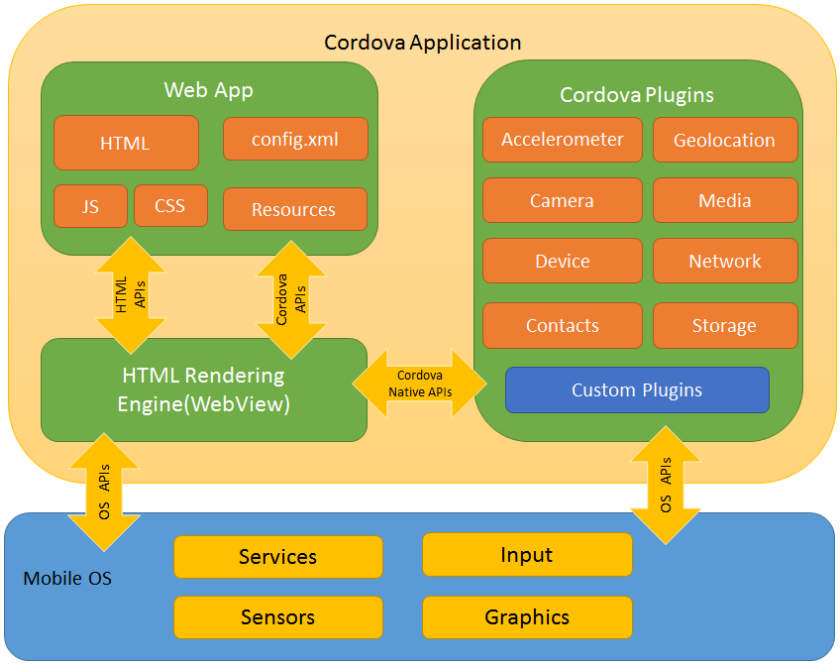

Abbildung 2.2: Funktionsweise von Cordova in einem Schaubild. [4]

für alle unterstützen Betriebssysteme und können mit entsprechenden Berechtigungen auf die Hardware-Komponenten zugreifen. Die WebView stellt also der Web-Applikation eine API zur Verfügung, um Plugins zu bedienen. Die Plugins selbst kommunizieren dann letztendlich mit den Hardware-Komponenten.
Eine weitere Möglichkeit zur Kommunikation mit Hardware-Komponenten bieten die HTML5-APIs, die auch in mobilen Browsern funktioniert. Dabei kann die WebView, wie auch ein Browser, auf einige Hardware-Komponenten zugreifen ohne ein Plugin zu verwenden. Als Beispiel kann auf das Mikrofon über die HTML5 Web-Audio-API zugegriffen werden, nachdem eine Berechtigung erteilt wurde. Jedoch ist ein Zugriff auf das Filesystem nicht standardmäßig mit einer der HTML5-APIs möglich. Diese Kommunikation muss über ein entsprechendes Plugin, in diesem Fall das Cordova *File Plugin*, erfolgen.

Die längerfristige Entwicklung deutet aber darauf hin, dass sich die HTML5-APIs weiterentwickeln und in Zukunft mehr Möglichkeiten bieten, wie zum Beispiel auch den Zugriff auf das Filesystem eines mobilen Gerätes.

3

Mobile App Frameworks

Die Applikation soll mit Hilfe von verschiedenen hybriden Mobile App Frameworks umgesetzt werden. Hybride Apps sind Web-Applikationen mit der Möglichkeit auf Hardwarekomponenten des Endgerätes zuzugreifen. Sie können auf verschiedenen Endgeräten und auf verschiedenen Betriebssystemen laufen. Dabei setzen hybride Apps auf Webtechnologien wie HTML5, CSS3 und Javascript und laufen dann in einer Art Browser, einem sogenannten Webview. Abhängig von der jeweiligen Plattform des Endgerätes läuft die Applikation beispielsweise bei Geräten mit Android als Betriebssystem in einem Chromium Webview.
Hybride Mobile Apps funktionieren dann auf mehreren Plattformen, sind aber genau genommen nicht *cross plattform*. *Cross plattform* Applikationen teilen sich zwar über mehrere Plattformen einen Großteil des geschriebenen Codes, aber die Benutzeroberfläche wird für jede Plattform durch nativen Code zusammengebaut. Dafür bieten einige Cross-Plattform-Frameworks auch Möglichkeiten, die Benutzeroberfläche betreffenden Code, zwischen den Plattformen zu teilen. Ein solches Cross-Plattform-Framework ist Xamarin. Es bietet mit seinem Feature *Xamarin.Forms* eine spezielle XAML-Syntax für graphische Elemente, die dann für jede Plattform zur Laufzeit das XAML in nativen Code umwandelt. Ein Beispiel für ein Cross-Plattform-Framework ohne zusätzliche XAML-Syntax ist React Native.
In dieser Arbeit werden die hybriden Mobile App Frameworks **Meteor**, **Ionic 2** und **Crosswalk** untersucht.

3.1 Meteor

Meteor ist ein Javascript Web-Framework für Webanwendungen und mobile Applikationen. 2011 wurde Meteor unter dem Namen *Skybreak* veröffentlicht und bekam in 2012 seinen jetzigen Namen. Im Gegensatz zu vielen hybriden Frameworks bietet Meteor die Möglichkeit zu jeder Applikation einen eng verknüpften Server zu programmieren. Zu den wichtigsten Bestandteilen von Meteor gehören *MongoDB* als Datenbank und das DDP (**D**istributed **D**ata **P**rotocol, welches zur Kommunikation zwischen der Client-Applikation und dem Server dient. Mobile Anwendungen werden durch die Unterstützung von Cordova (siehe 2.2) erstellt.

Meteor verfolgt bei Anwendungen vier Grundprinzipien [5]. Meteor Applikationen benutzen nur **eine Programmiersprache** in allen Meteor Umgebungen. Das bedeutet sowohl in der mobilen und webbasierten Client-Applikation, als auch auf Seiten des Servers wird Javascript verwendet. Meteor verfolgt das **data on the wire**-Konzept, was mit Hilfe des DDPs umgesetzt wird. Das bedeutet, dass zwischen Client-Applikation und Server nur Daten gesendet werden und keine vollständigen HTML-Seiten.
Mit dem eigenen Package-System, für das alle Benutzer eigene autonome Programmteile erstellen und der Allgemeinheit zur Verfügung stellen können, verfolgt Meteor die Strategie **embrace the ecosystem** und bindet damit aktiv die Community in die Plattform ein. Das letzte Prinzip besteht in **full stack reactivity**. Dadurch ist es mit Meteor möglich, den aktuellen Zustand der Applikation mit minimalem Aufwand im User Interface darzustellen.

3.1.1 Architektur

Meteor Applikationen bestehen aus mehreren Ebenen und Komponenten, die miteinander kommunizieren. Die zwei Hauptebenen sind die Client-Ebene und Server-Ebene. Von der Server-Ebene können Datenbanken und externe REST-Services angesprochen angesprochen werden. Im Schaubild 3.2 ist die Architektur einer Meteor-Applikation veranschaulicht.

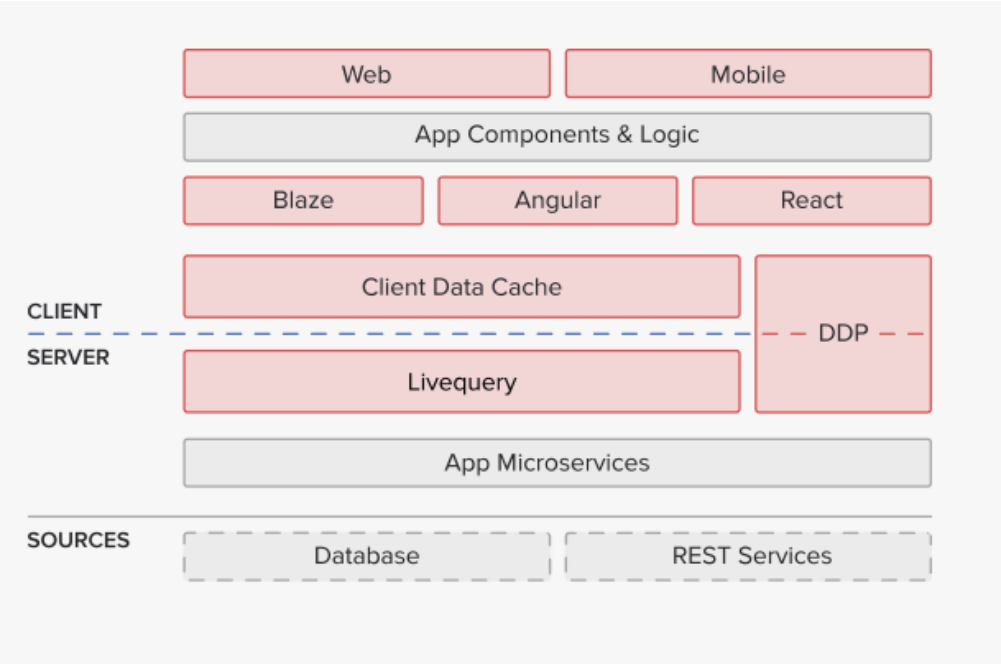

Abbildung 3.1: Architektur einer Meteor-Applikation. Entnommen aus [6]

User Interface

Meteor unterstützt unabhängig von der Zielplattform verschiedene UI-Frameworks. Während in frühen Versionen *Blaze* als einziges Framework zur Auswahl stand, sind mittlerweile auch das von Google entwickelte *AngularJS* und das von Facebook entwickelte*React* eine Option.

Blaze

BlazeJS ist eine *Rendering Library* für User Interfaces und wurde 2011 als Teil von Meteor entwickelt. Blaze besteht zum einen aus der Template-Engine *Spacebars*, einer Weiterentwicklung von *Handlebars* (siehe [7]). Spacebars erweitert einfaches HTML um reaktive Elemente durch spezielle Syntax. Im Beispiel 3.1 ist ein Template mit dem Namen **example** definiert. Dabei wurden bekannte HTML-Tags wie **<h1>** durch Elemente mit doppelt geschweiften Klammern erweitert.

```
1 <template name="example">
2   <h1>{{title}}</h1>
3   <div>
4       {{#if birthday}}
5         Happy Birthday!
6       {{/if}}
7   </div>
8 </template>
```

Listing 3.1: Beispiel für durch Spacebars erweitertes HTML

Dieses Template wird dann von der Template-Engine auf die erweiterte Syntax durchsucht und ersetzt die Spacebars-Elemente durch entsprechendes valides HTML. Die Logik für die Spacebars-Elemente wird in Javascript in Form von sogenannten Helpers definiert (siehe Beispiel 3.2). Für das Template **example** werden zwei Helper definiert, die von der Template-Engine dann zum Ersetzen der Spacebars-Elemente verwendet werden. Dabei kann ein Helper nicht nur durch einfache Variablen, sondern auch durch eine komplexe Funktion definiert werden. Zur Laufzeit wird so das Template mit Hilfe der Helper kompiliert.

```
1 Template.example.helpers({
2   title : "Welcome!",
3   birthday : function(){
```

```
4       return userHasBirthday(); //diese Methode sei an anderer
5     }                           //Stelle definiert.
6   });
```

Listing 3.2: Beispiel für Spacebars-Helper

Die Template-Engine kompiliert aber nicht nur einmalig, sondern jedes mal wenn sich die Abhängigkeit eines Helpers ändert. Man nennt diese Strategie **data-binding**. Dabei rendert die Template-Engine Spacebars-Elemente neu, wenn sich die Daten, von denen der Helper abhängig ist, ändern.

Angular

AngularJS ist ein Framework zum Erstellen von Client-Applikationen und wurde 2009 von Google veröffentlicht. 2016 wurde dann Angular (auch bekannt als Angular 2) veröffentlich, als neue Version mit überarbeiteten Konzepten und einer neuen Architektur. Meteor integriert sowohl AngularJS, als auch das aktuelle Angular. Dieses Unterkapitel behandelt Angular 2.

Als Programmiersprache benutzt Angular TypeScript, einer Erweiterung von Javascript durch Klassen, Interfaces und besserer Typsicherheit für große Applikationen. Die genauen Spezifikationen von TypeScript sind hier [8] nachzulesen.
Angular-Applikationen bestehen aus sogenannten **Components**, die wiederum aus einer TypeScript-Klasse und einem dazugehörigen HTML-Template bestehen. Components sind modulare Klassen, die von anderen Components importiert und verwendet werden können, wodurch sich eine Hierarchie entwickelt. In dieser Hierarchie können Components miteinander kommunizieren. Im Codebeispiel 3.3 sieht man einen sehr einfach aufgebauten Component. Das HTML-Template ist als Teil des Components definiert und weißt Ähnlichkeiten zu den Blaze-Templates auf, da das Angular-Template ebenfalls spezielle Syntax verwendet, die beim Kompilieren für **data-binding** verwendet wird. Optional können Templates auch in externe Dateien ausgelagert werden und im

Component referenziert werden.

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-app',
  template: `<h1>Hello {{name}}</h1>`
})
export class AppComponent { name = 'Angular'; }
```

Listing 3.3: Beispiel für ein Angular-Component, entnommen aus [9]

In den Angular-Templates können **ng-directives** verwendet werden. Diese gehören zur Angular-spezifischen Syntax und werden verwendet, um komplexe Strukturen darzustellen, wie zum Beispiel das mehrmalige Rendern des gleichen Codeteils durch eine Schleife. Schleifen können in Angular durch die **ngFor-directive** gebaut werden, was in Codebeispiel 3.4 dargestellt ist. Der Component hat in dem Fall einen Array von Spielern als Membervariable *players* und benutzt diesen Array in seinem Template, indem er in jeder Schleifeniteration einen Spieler durch ein Listenelement mit Spielername und seiner Punktzahl darstellt.

```
<ul>
  <li *ngFor="let player of players">
    <h4>{{player.name}}</h4>
    <p>{{player.score}}</p>
  </li>
</ul>
```

Listing 3.4: Beispiel für die ngFor-directive in Angular

React

React ist ebenfalls eine Javascript-Bibliothek zum erstellen von reaktiven User Interfaces und wurde 2013 von Facebook veröffentlicht. Ähnlich zu Angular verfolgt React den Ansatz, Applikationen in modulare Components einzuteilen. React wird in purem Javascript oder in JSX programmiert. In JSX wird Javascript mit eingebetteter XML-Syntax kombiniert. Das bedeutet, im Javascript-Code können XML-Tags verwendet werden, ohne diese in einen String zu schreiben. Für JSX-Files gibt es momentan noch keinen weitreichenden Browser-Support, weshalb es verschiedene Kompilatoren gibt, die JSX in pures Javascript umwandeln.

React-Components bestehen aus einem Konstruktor, einem State und einer Render-Funktion. Der State spiegelt den momentanen Zustand des Components wieder in Form von Variablen. Der Konstruktor kann verwendet werden, um den Component in einen initialen State zu versetzen. Die Render-Funktion ist dem Template in Angular ähnlich, denn sie enthält HTML und kann durch spezielle Syntax auch Variablen des States darstellen. Wenn sich der State verändert, wird abhängiges HTML aktualisiert.

```
1  class Message extends React.Component {
2    constructor(props) {
3      super(props);
4      this.state = {message: "Hello World!"};
5    }
6    updateMessage(){
7      this.setState((prevState) => ({
8        message: "You just clicked!"
9      }));
10   }
11   render() {
12     return (
13       <h1 onClick={this.updateMessage} >{this.state.message}</h1>
```

```
14      );
15    }
16  }
```

Listing 3.5: Beispiel für einen React-Component in JSX

Das Codebeispiel 3.5 enthält einen React-Component, der in seinem Konstruktor seinen State mit einem String initialisiert, dieser wird in der Render-Funktion in einen Headline-Tag eingebettet. Außerdem wird dem Tag ein Event-Listener zugewiesen, der bei Klick die Methode *updateMessage* des Components ausführt. Diese Methode aktualisiert den State des Components, was dazu führt, dass sich der Überschrift ändert.

```
1
2  render() {
3    return (
4      <div>
5        <Message example="true" />
6      </div>
7    );
8  }
```

Listing 3.6: Beispiel für Hierarchien zwischen Components

Components können in React auch ineinander geschachtelt werden, um so Hierarchien zu erschaffen. Codebeispiel 3.6 zeigt die Render-Funktion eines Component, die den Component aus Beispiel 3.5 verwendet, indem ein XML-Tag mit dem Namen dieses Component eingefügt wird. Hierbei entsteht eine Hierarchie mit dem *Message*-Component als Kind-Component. Dabei können wie in dem Beispiel mit Hilfe von Attributen auch Abhängigkeiten übergeben werden, auf die dann vom Kind-Component über *this.props* zugegriffen werden kann.

Aufteilung in Client und Server

Meteor-Applikationen teilen sich einen Client- und in einen Serverteil auf. Weiterhin teilen sich Client und Server auch große Teile des Codes. Wie in Codebeispiel 3.7 gezeigt wird, kann mit *Meteor.isClient* und *Meteor.isServer* festgelegt werden, wo welcher Code laufen soll. Codebereiche, für nichts festgelegt wird, werden sowohl auf dem Client, als auch auf dem Server ausgeführt.
Die gesamte Applikation wird bevor sie läuft, kompiliert. Bei der Kompilation teilt Meteor den Code entsprechend in die Client- und in die Serverapplikation auf. Außerdem ist auch möglich durch die Ordnerstruktur festzulegen, wo welcher Code laufen soll. Dafür kann ein /**client**- und ein /**server**-Ordner in der Applikation angelegt werden.

```
1 if(Meteor.isClient){
2   //client only
3 }
4 if(Meteor.isServer){
5   //server only
6 }
7 //shared code
```

Listing 3.7: Verwendung von Client- und Server-Code

Datenbanken

Neben der MongoDB als Teil der Serverapplikation gibt es auch auf der Seite des Clients eine Datenbank. Diese Client-Datenbank nennt sich Minimongo und gleicht bezogen auf die Funktionalität einer MongoDB. Minimongo ist in Javascript geschrieben und kann JSON-Objekte abspeichern. Das primäre Ziel dieses Konzepts, ist es Datensätze der Server-Datenbank in der Client-Applikation zwischen zu speichern.

Meteor nutzt zum Datenaustausch zwischen Client und Server das **publish-subscribe pattern**. Das *publishing* findet auf dem Server statt, wobei ausgewählte Datensätze zur

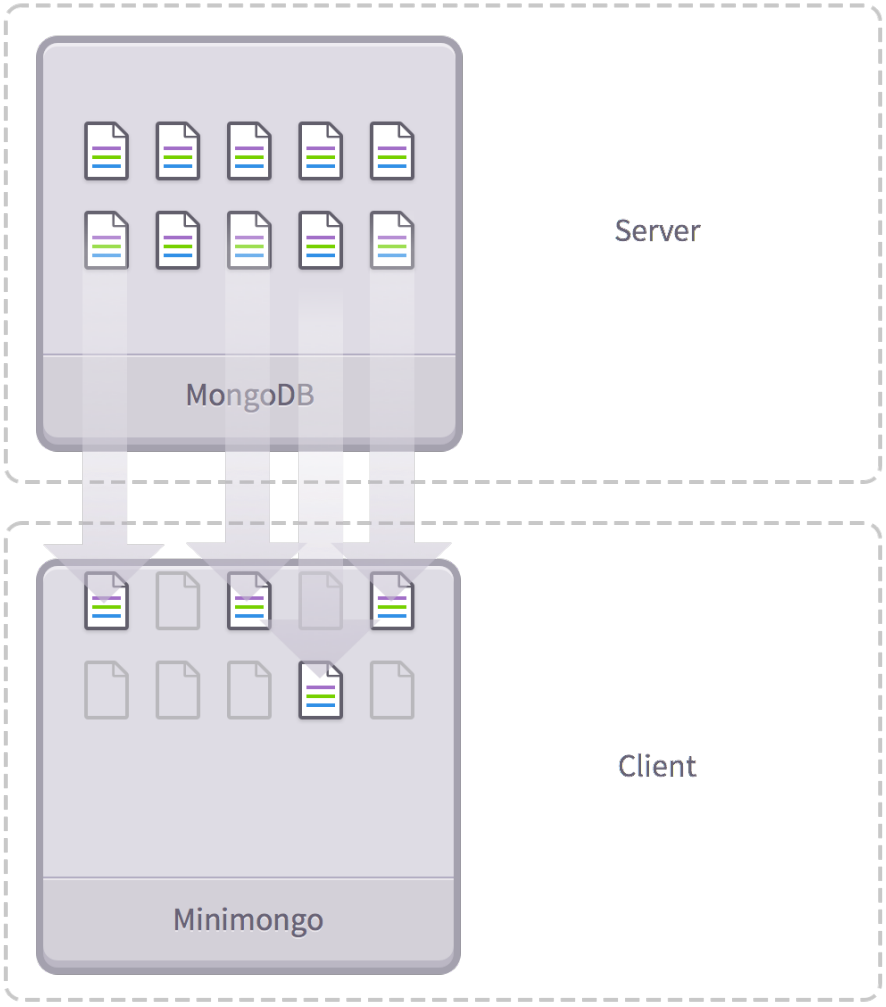

Abbildung 3.2: Das **publish-subscribe pattern** in Meteor. Entnommen aus [10]

Verfügung gestellt werden. Zu diesen Datensätzen können Clients *subscriben*, das heißt Datensätze werden angefordert. Falls der Client sich mit den nötigen Rechten identifizieren kann, werden nun alle angeforderten Daten in die Minimongo des Clients eingefügt. Nach diesem Vorgang können die Daten beispielsweise zur Darstellung verwendet werden. Wenn sich Datensätze auf dem Server ändern, werden durch Subscriptions betroffene Clients informiert. Dadurch wird die Minimongo Datenbank des Clients stetig auf einem aktuellen Stand gehalten.

Da Meteor als Datenbank MongoDB benutzt, werden Datensätze in *Collections* abgespeichert. Eine Collection mit dem Namen SStore"kann mit dem Befehl **Store = new Mongo.Collection('store');** erschaffen werden. Auf diesem Objekt können dann Operationen zum finden, ändern und löschen ausgeführt werden. Auf dem Client sind Collections inital immer leer, auch wenn eigentlich Datensätze enthalten sind. Mit Hilfe der Methoden *publish* auf dem Server und *subscribe* wird die Collection auf dem Client mit Daten befüllt.

Server und Kommunikation mittels DDP

Die Client-Applikation und der Server sind in Meteor-Applikationen eng miteinander verbunden, da sie eine beständige Kommunikation miteinander führen. Zur Kommunikation benutzt Meteor das auf Basis von Websockets eigens entwickeltes Protokoll, das **D**istributed **D**ata **P**rotocol [11]. DDP-Nachrichten sind JSON-Objekte und werden über Websockets zwischen Client und Server ausgetauscht. Dabei sind im DDP bestimmte Nachrichtentypen festgelegt, beispielsweise die *ping*-Nachricht, die der Server nutzt, um festzustellen, ob die Verbindung zum Client noch besteht. Der Client würde dann mit einer *pong*-Nachricht antworten, um zu bestätigen.

Das Protokoll soll in erster Linie zwei Aufgaben erledigen. Zum einen soll es vom Client ausgehende Aufrufe an dem Server, sogenannte *remote procedure calls* ermöglichen. Zum anderen soll es das *publish-subscribe pattern* umzusetzen, das heißt, wie in

Abschnitt 3.1.1 erklärt, dem Client Datensätze zukommen zu lassen und diese aktuell zu halten.

Meteor Methods (Remote Procedure Calls)

Meteor Methods sind Funktionen, die auf dem Server definiert werden. Eine solche Methode ist ein *key-value*-Paar, wobei der *key* ein eindeutiger Name und die *value* eine Funktion ist. Diese Methoden können vom Client mittels **Meteor.call("method_X");** aufgerufen werden, wobei "method_xëin Platzhalter für den Namen der Methode meint. Sie dienen also als Mittel zur Kommunikation vom Client zum Server zu einem beliebigen Zeitpunkt. Für diese sogenannten *Remote Procedure Calls* wird das DDP benutzt.

3.1.2 Mobile Anwendungen

3.2 Ionic 2

3.3 Crosswalk

4

Anforderungen

Vor der Umsetzung einer Applikation, müssen die Anforderungen festgelegt werden. Anforderungen kann man in zwei Kategorien unterteilen. Funktionale Anforderungen betreffen genaue Leistungen, die die Anwendung erbringen soll. Nicht-funktionale Anforderungen beschreiben, wie gut die Leistung erbracht werden soll.

4.1 Funktionale Anforderungen

In diesem Abschnitt werden die funktionalen Anforderungen an die Applikation definiert.

FA01 Erste Anforderung
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua.

FA02 Zweite Anforderung
At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet.

4.2 Nicht-Funktionale Anforderungen

Neben Anforderungen funktionaler Art, gibt es auch gewisse Qualitätseigenschaften und Rahmenbedingungen, die die Applikation erfüllen soll. Diese nicht-funktionalen Anforderungen sind im Folgenden definiert.

- **Zuverlässigkeit**
  Die Applikation sollte keine fehlerhaften Zustände annehmen oder sogar abstürzen.
- **Wartbarkeit**
  Es soll nachträglich möglich sein, die Applikation zu verändern und um Funktionen zu erweitern.
- **Sicherheitsanforderungen**
  Sicherheitsrelevante Daten sollen verschlüsselt übertragen werden und vor dem Zugriff Dritter geschützt sein.
- **Benutzbarkeit**
  Der Benutzer soll durch ein verständliches User-Interface schnell die Funktionen der Applikation kennenlernen und mit diesen umgehen können.
- **Design ("Look & Feel")**
  Durch ansprechende Gestaltung der Applikation, soll sich der Benutzer im Umgang mit der Applikation wohl fühlen.

Literaturverzeichnis

[1] Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. dissertation, UNIVERSITY OF CALIFORNIA, IRVINE (2000)

[2] Masse, M.: REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces. O'Reilly Media, Inc. (2011)

[3] Crockford, D.: The JavaScript Object Notation (JSON) Data Interchange Format . https://tools.ietf.org/pdf/rfc7159.pdf (2014) Zugriff: 2017-01-05.

[4] Foundation, A.S.: Overview and Architecture of Cordova Applictions. https://cordova.apache.org/docs/en/6.x/guide/overview/index.html (...) Zugriff: 2017-01-04.

[5] Group, M.D.: What is Meteor? (https://guide.meteor.com/#what-is-meteor) Zugriff: 2017-01-10.

[6] Agüero, F.: 5 reasons why I choose React-Meteor for my projects. (http://fjaguero.com/blog/5-reasons-why-i-choose-react-meteor-for-my-projects/) Zugriff: 2017-01-11.

[7] Contributers, V.G.: Handlebars. (http://handlebarsjs.com/) Zugriff: 2017-01-12.

[8] Bierman, G., Abadi, M., Torgersen, M. In: Understanding TypeScript. Springer Berlin Heidelberg, Berlin, Heidelberg (2014) 257–281

[9] Inc., G.: Angular Quickstart. (https://angular.io/docs/ts/latest/quickstart.html) Zugriff: 2017-01-13.

[10] Greif, S.: Understanding meteor publications & subscriptions. https://www.discovermeteor.com/blog/understanding-meteor-publications-and-subscriptions/ (2014) Zugriff: 2017-01-18.

[11] Contributers, V.G.: DDP Specification. (`https://github.com/meteor/meteor/blob/master/packages/ddp/DDP.md`) Zugriff: 2017-01-18.

Abbildungsverzeichnis

Tabellenverzeichnis

Name: Julius Friedrich Matrikelnummer: 841963

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Julius Friedrich